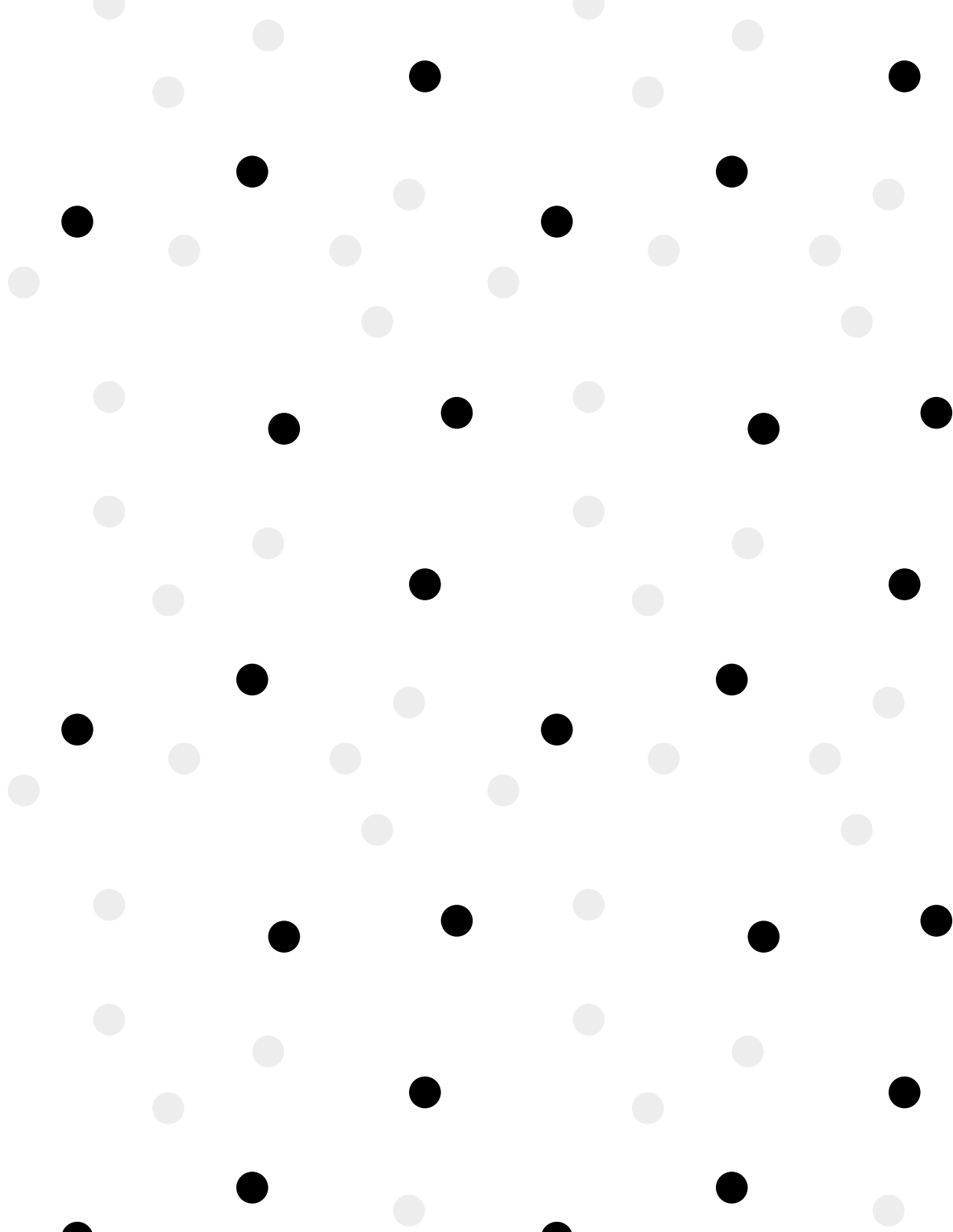


Session 1 Intro to C

Program Entry Points (main function)

→ program is launched / executed by OS / bash shell

→ calling main: transfers control flow to the program → OS after execution

→ no parameter, int type

Return type: data type that is returned to the caller (int, void, bool ...)

identifier: name of the function (int main(void))

parameter list: seperated by comma, with its own data type

Void: no return value / no parameter ← use void to tell if no param

void ... ( ) { } .... (void)

Function body: { } ← within this

return: return corresponding data type to the caller (must match)

Data type:

Static typing: all types must be known when program compiles
type of identifiers cannot change

Data: stored in memory as a sequence of bits (1, 0)
8 bits = one byte

4 built-in data type: int, char, bool, float

Int: 32 bit (4 bytes) between INT_MIN and Int_max (-2147483648, 2147483647)

Bool: 32 bit (4 bytes) true / false

Note: no type-based run-time errors

do not add a leading zero → evaluate as OCtal (base 8), 0x → hexadecimal (base 16)

Operators

Arithmetic operator: like normal math operations (avoid using % with negative #)

Bool: || (OR) (All false = false, otherwise true) && (and) (all true = true, otherwise false) ! (not, negation)

== (double equality sign) != (not equal) <, <=, >, >= (comparison)

Precedence	Operator	Description	Associativity
1	++ --	Postfix (suffix) increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type) {list}	Compound literal	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and Unary minus	
	! ~	Logical NOT and Bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement	

Precedence	Operator	Description	Associativity
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise shift left and bitwise shift right	
6	< <= => >	Relational comparators	
7	== !=	Relational equality and inequality	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional	Right-to-left

Precedence	Operator	Description	Associativity
14	=	Assignment	Right-to-left
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	&	Comma	Left-to-right

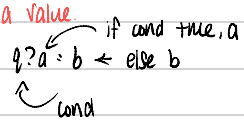
Conditionals:

if, else, else if

Note: if not void → all if else need return statement if necessary

if does not produce a value.

(?:) produces a value



Tracing:

Tracing functions:

trace_int( ) trace_bool( ) ....

[main.c | main | 8] >> my_sqr(4) => 16

file name | function name | Line # | expression | return value / result

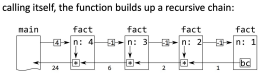
Recursion

Recursion a function called itself

Simple recursion

```
// fact(n) calculates the factorial of n.
int fact(int n) {
  if (n == 1) { // base condition
    return 1;
  } else {    // recursive condition
    return n * fact(n - 1);
  }
}
```

By calling itself, the function builds up a recursive chain:



Accumulative:

```
int fact_worker(int n, int acc) { // worker function
  if (n == 1) {                   // base condition:
    return acc;                   //   return accumulator
  } else {                        // recursive condition:
    return fact_worker(n - 1,     //   recursive call, with
                       n * acc);  //   recursive argument
                                  //   one step closer to
  }                               //   base condition and
}                                 //   accumulator combined
                                  //   with current value

// fact(n) calculates the factorial of n.
int fact(int n) {
  return fact_worker(n, 1); // accumulator initialized with
}                           //   base value
```

Assert

assert testing-harness assert(cond)

Intro to Imperative C

procedure Programming subtype of imperative paradigm
sequence of instructions executed

```
int main(void) {
  trace_int(1 + 1);    // First, do this ...
  assert(3 > 2);       // ... then do this ...
  return EXIT_SUCCESS; // ... and then do this.
}
```

Side effect occurs when the state of something changes
↳ some data at the moment

procedure uses side effect. function not

Type: Input/Output (I/O) or Mutation (memory modification / variables)

I/O

I/O how program interact with real world

Input: System output (Files, network data)
User input
Sensor input (camera, GPS...)

Output Visual (screen)
Auditary (sounds)
Tactile (vibration)

printf("") C function for printing out the text "\n" new line
↳ format string

Additional argument printf("...%d", 2+2) additional arg
↳ output is int (placeholder)

output % → 2 %% backslash: "\\"

% beginning of a placeholder '' char " " string \ new char with meaning

Functions with side effects

Documenting Side Effects

The `printf` function has a side effect: it changes the output stream, i.e., the state of the system, **permanently** (and irrevocably). By calling `printf`, the function `noisy_sqr` inherits its side effect.

```
// noisy_sqr(n) squares n and writes a confirmation to output.
// effects: produces output
int noisy_sqr(int n) {
  printf("I'm squaring %d\n", n);
  return n * n;
}
```

Clearly communicate such a side effect in the `effects`–section!

trace / assert → not side effect

Functions with Side Effects: Terminology

```
int noisy_sqr(int n) {
  printf("I'm squaring %d\n", n);
  return n * n;
}

int main(void) {
  trace_int(noisy_sqr(7));
}
```

For `noisy_sqr`, you should say: if you **pass** 7 to `noisy_sqr`, it **outputs** a message and **returns** 49.

void function returns nothing

Variable

System State storing info in memory, associating with an identifier

Variables store mutable state info

need type(int ...?), the identifier (name), initial value

=, ; complete the syntax

Mutation change the value of a variable

is a side effect

m = m + 1;

Assignment operator LHS → name RHS → expression

Not symmetric: y=x != x=y

```
int n = 5; // initialization syntax
n = 6;     // assignment operator
```

Note: Always initialize although it is ok to not

+=, *=, /=, -=, %=, ++, --, (++t, t++ are different)

Const use when appropriate

Stores data that is immutable

Global Data defined outside functions **Local Data** defined within the function

Data Scope region of code where it is "accessible" or "visible"

For global data, the scope is anywhere **below** its definition.

```
int main(void) {
  printf("%d\n", course);
}

int course = 136;
```

> use of undeclared identifier 'course'

Local data has **block scope**. Its scope extends from its definition to the end of the block it is defined in.

```
int foo(int n) {
  // ...
  int a = 42;
  // ...
  if (n > 0) {   // beginning of the block where b is defined
    // ...
    int b = 136; // b is defined
    // ...
  }              // end of the block where b is defined
  // ...
}
```

in scope for the entire functions

Note: mutate global variable is a side effect, but local is not (pure) (and param) (impure)

I/O

read_int() read next integer or READ_INT_FAIL

↳ next integer could not successfully read

↳ end of input

read from stdin window

once the value is read → not read again

.in, .expect testing files

↳ input file → expected output files

Text Input: Reading Input

We can use a recursive approach to read all integers from input.

```
// read_all_int() reads all integers from input and ...
// effects: reads from input
void read_all_int(void) {
  int input = read_int();
  if (input == READ_INT_FAIL) { // base condition
    return;
  } else {                    // recursive condition
    // do something with input
    read_all_int();
    // do something with input
  }
}
```

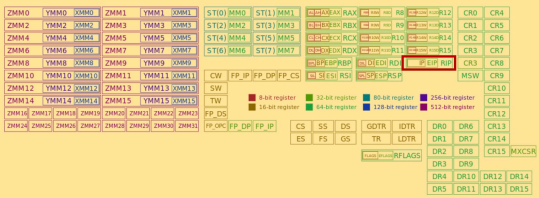
C Control Flow and Memory

Control Flow

Control flow model how programs are executed

Type: Function calls ①
Conditions ②
Iterations ③

In hardware, the location (register) is known as the *program counter* or *instruction pointer* and contains the memory address of the current instruction.



(More on this in CS 241).

① When the function is called, the program location "jumps" to the start
returns → back to caller

Function Calls: Return

The control flow statement `return` changes the program location to go back to the most recent calling function: the control flow returns from the *callee* back to the *caller*.

② The syntax of `if` is:

```
if (predicate)
  statement
```

where the `statement` is only executed if the `predicate` evaluates to `true`:

```
if (n < 0) {
  printf("n is less than zero\n");
}
```

Remember: the `if` statement does not yield a value. It only controls the flow of execution.

Statement may/may not executed
only one will be executed (or stm none)

③ 1. While loop

```
while (predicate)
  statement / compound statement
```

loop can be nested

2. do-while

```
do {
  compound_statement
} while (predicate);
```

statement is always executed at least one

break Jumps out the innermost loop

continue skips over the rest and continues with the loop

3. for loop

```
for (setup statement; predicate; update statement) {
  body statement
}
```

Memory

one bit 0 / 1 **byte**: 8 bits, smallest accessible unit of memory

address position in memory

addr	data (1 byte)
0x00	136
0x01	255
0x02	42
0x03	94
0x04	33
...	...
0xFE	127
0xFF	192

Structure of Memory: Defining Variables

Whenever a variable is defined, C
- reserves sufficient space in memory to store the variable,
- associates the location (i.e., address) of that space with the identifier,
- writes the initial value of the variable to that location (i.e., address).

size of yields the number of bytes required to store a type (operators)

Integer Limit Int_min, Int_max

Integer Overflow exceeds limits

Additional Type

char store single characters (integer value based on ASCII)

'' syntax for char

%c printf placeholder for char

Input: read_char (READ/IGNORE_Whitespace)

Invisible	Punct.	Digits	Punct.	Upper-case Letters	Punct.	Lower-case Letters	Punct.
0 \0	32 ␣	48 0	58 :	65 A	78 N	91 [	97 a	110 n	123 {
	33 !	49 1	59 ;	66 B	79 O	92 \	98 b	111 o	124 |
	34 "	50 2	60 <	67 C	80 P	93]	99 c	112 p	125 }
	35 #	51 3	61 =	68 D	81 Q	94 ^	100 d	113 q	126 ~
	36 $	52 4	62 >	69 E	82 R	95 _	101 e	114 r	
	37 %	53 5	63 ?	70 F	83 S	96 `	102 f	115 s	
	38 &	54 6	64 @	71 G	84 T		103 g	116 t	
	39 '	55 7		72 H	85 U		104 h	117 u	
	40 (	56 8		73 I	86 V		105 i	118 v	
	41)	57 9		74 J	87 W		106 j	119 w	
10 \n	42 *			75 K	88 X		107 k	120 x	
	43 +			76 L	89 Y		108 l	121 y	
	44 ,			77 M	90 Z		109 m	122 z	
	45 -								
	46 .								
	47 /								

Float (floating point #)

represent real number

%f float placeholder

Structures

```
struct posn { // name of the structure
  int x;      // field type and identifier
  int y;
};            // closing semicolon
```

name

don't forget

initialization - `struct posn p = {.y = 4, .x = 3};`

- field axess operator p.x => 3 p.y => 4

printf: axess all field one by one, not the structure it self

Sections of Memory

Code
Global Constants
Global Variables
↓↓↓ Heap ↓↓↓
↑↑↑ Stack ↑↑↑

← code in text-editor (convert to machine code then compile)

← avaible throughout the entire execution of the program

← all stacks frames

Intro to Pointers

Pointer Basics

Address Operator

Address Operator

```
int val = 136;
printf("Value of val: %d\n", val);
printf("Address of val: %p\n", &val);

> Value of val: 42
> Address of val: 0xC0
```

id	type	addr	data (4 bytes)
		...	...
val	int	0xC0	136
		...	...
		...	...
		...	...
		...	...

The `printf` placeholder to display an address (in hex) is `%p`.

Pointer types

A pointer is defined by placing a star * after the type. The * is part of the type syntax, **not** the identifier.

```
int val = 136;
int *pval = &val; // the pointer pval "points at" val
```

The type of `pval` is of type `int`-pointer which is written as `int *`.

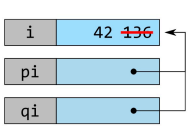
For each type (e.g., `int`, `char`, `struct posn`) there is a corresponding pointer type (e.g., `int *`, `char *`, `struct posn *`).

dereference operator

aliasing multiple pointers to the same data

```
int i = 136;
int *pi = &i;
trace_int(i);
trace_int(*pi);

int *qi = &i; // same as: int *qi = pi;
*qi = 42;     // *qi => i
trace_int(i);
```



Pointer type parameter

Pass by value:

```
int inc(int x) {
  return x + 1;
}

int main(void) {
  int x = 5;
  trace_int(x);
  x = inc(x);
  trace_int(x);
 }
```

Pass by reference: → allows a function to write into the stack frame of the caller

passing the address of x mutates the value of x

```
void inc(int *px) {
  *px += 1;
}

int main(void) {
  int x = 5;
  trace_int(x);
  inc(&x);
  trace_int(x);
}
```

In the *pass by value* version, a **copy of an argument** is passed to a function. In the *pass by reference* version, a **reference** (pointer) to the stack-frame of the caller function is passed.

Returning an Address

```
1.  int *bad_idea(int n) {
2.    return &n;
3.  }
4.
5.  int main(void) {
6.    int *ptr = bad_idea(136);
7.    trace_ptr(ptr);
8.   ▸trace_int(*ptr);
9.  }
```

```
> ptr => 0xD0
> Error: stack-use-after-return
```

```
=========================
bad_idea:
  n:       136   [addr_1]
  r/a: main:6
-------------------------
main:
  ptr: addr_1
  r/a:     OS
=========================
```

scanf("%d", &input) reads int, stores in input
ignores whitespace

scanf(" %c", &input) reads char, with whitespace

scanf("%c", &input) ignore whitespace

Structure Type

passing structure

Note:

```
int *find_min(int *a, int len) {

  int *cur_min = a;<- since input is already a pointer


  for(int *i = a; i < a + len; ++i) {
    if (*i <= *cur_min) {
      cur_min = i;<- cur_min = its current location
    }
  }

  return cur_min;<- return its location
}


void selection_sort_descending(int a[], int len) {
  for (int i = len - 1; i >= 0; --i) {
    int temp = a[i];
    int *min = find_min(a, i);
    a[i] = *min;= a[ (location to min)]
    *min = temp;
  }
}
```

a[i] = *i

void reverse_arry_idx (int *arr, int len) {

arr[5] = {2, 4, 6, 0, 1}

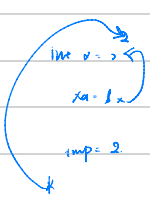
xp = &arr[4] p = 4 q = 2

xq = &arr[2]

*i = &a = 2.

temp = *i 2.

&arr[0] = *(a + len - 2



Modules

Modularizations

3 advantages: reusability, maintainability, abstraction

Reusability re-used by clients

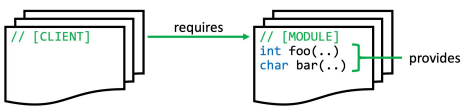
Abstraction understand functionalities, not implementation

Maintainability only have to fix bugs in module

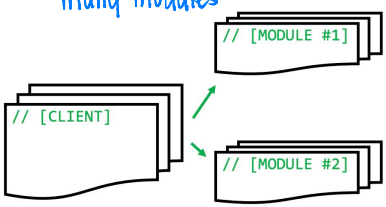
large program can be built from many modules

Terminology

It is helpful to think of a "client" that requires the functions that a module provides.



In practice, the client is a file that may be written by yourself, a co-worker, or someone else.

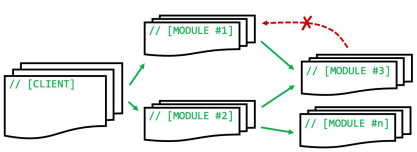


Terminology

Large programs can be built from many modules.

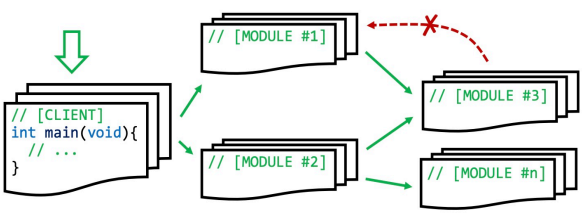
A module can be a client itself and require functions from other modules.

The module dependency graph should not have any cycles.



16

There must be a "root" that acts as the client. This is the program file that contains the `main`-function (i.e., the program entry point) and is "run".



17

Modules in C

Declaration introduces an identifier

Definition gives some content to an identifier (also contains declaration)

identifiers can be declared multiple times, but defined once

extern optional for declaration

Definition

consists of the signature and body

A function *declaration* (or function prototype) consists of the `extern` keyword followed by *signature* of the function.

```
extern int math_sqrt(int n);
```

Declaring and Defining Global Data

A global data (global variable or global constant) *declaration* consists of the `extern` keyword, the type, and the identifier.

```
extern const int PI;
extern int global_variable;
```

A global data *definition* consists of the type, the identifier, and the initializer. extern is optional

```
const int PI = 3;
int global_variable = -1;
```

Declaring and Defining Structures

A structure declaration consists of the `struct` keyword and the name (*incomplete declaration*), or the `struct` keyword, the name and the field declarations (*complete declaration*).

```
struct posn;
```

```
struct posn {
  int x;
  int y;
};
```

A structure *definition* consists of the `struct` keyword, the name, the identifier, and the initializer.

```
struct posn my_pos = {3, 4};
struct posn my_other_pos = {.x = 3, .y = 4};
```

We cannot use structures with incomplete declarations until we learn about using heap memory (Section 10).

23

.h extension only module extensions, documentation (header file)

```
// cs136-math.h

// PI: the ratio of a circle's circumference to its diameter
extern const int PI;

// math_sqrt(n) returns the square root of n, rounded
//   towards 0.
// requires: n must be non-negative
int math_sqrt(int n);
```

Note:
identifier must declared before use

The header-file provides the *interface* of the module.

.C extension (source file) contains module definition

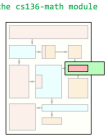
Standard Modules < > for standard, " " for non-standard

Black-box test client

For each module you design, it is good practice to create a **test client** that ensures the provided functions are correct.

```
// cs136-math-test.c: testing client for the cs136-math module
#include <assert.h>
#include "cs136-math.h"

int main(void) {
  assert(math_sqrt(0) == 0);
  assert(math_sqrt(1) == 1);
  assert(math_sqrt(3) == 1);
  assert(math_sqrt(4) == 2);
  assert(math_sqrt(8) == 2);
  assert(math_sqrt(9) == 3);
  assert(math_sqrt(999999) == 9999);
  assert(math_sqrt(1000000) == 10000);
}
```



Do NOT add a main function to your implementation (e.g., `cs136-math.c`). Create a new test client with a main function.

White-box test client

There may be internal functionality that (external) black-box test clients cannot test. These may include implementation-specific tests and tests for module-scope functions.

In these circumstances, you can provide a `test_module_name` function within your module that tests all internal functionality of your module.

Interface vs Implementation

↳ .h file ↳ .c file

- information hiding secure and flexible
  - ↳ allows changes without affecting client
  - ↳ prevents clients from direct access to the data

Opaque Structure in C

The level of information hiding can be further increased by providing only incomplete declarations for structures.

```
struct posn;
```

```
struct posn {
  int x;
  int y;
};
```

When providing an incomplete structure declaration, clients have no information about or access to the structure's fields.

only pointers to an opaque structure is defined

```
struct posn my_posn;      // INVALID
struct posn *my_posn_ptr; // VALID
```

Transparent Structure

all defined in C

Abstract Data Types

provides certain behaviour when storing data

As the client, if you have a data structure, you **know** how the data is structured and you can **access the data directly** in any manner you desire.

With an ADT, the client does **not** know how the data is structured and can only access the data through the *interface* provided by the ADT.

Collections

ADT, store an arbitrary amount of data

Stacks, Queues, Lists/Sequences Trees Graphs Sets

Stact ADT

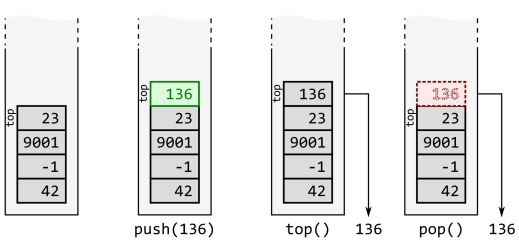
items are pushed onto the top of stack

popped off the top exists (last in, first out)

operation

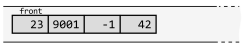
Typical stack ADT operations:

- **push**: adds an item to the top of the stack
- **top**: returns the item at the top of the stack
- **pop**: removes the item from the top of the stack and returns it
- **empty?**: determines if the stack is empty or not



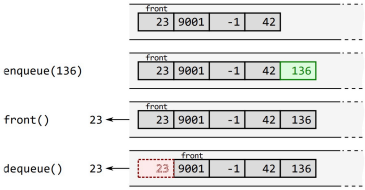
Queue ADT

A queue is like a lineup at your favourite beverage vendor. In a queue ADT, new items are added to the back of the line, and items are removed from the front of the line. While a stack exhibits *LIFO* behaviour, a queue exhibits *FIFO* behaviour (first in, first out).



Typical queue ADT operations:

- **enqueue**: adds an item to the end of the queue
- **front**: returns the item at the front of the queue
- **dequeue**: removes the item from the front of the queue and returns it
- **empty?**: determines if the queue is empty or not

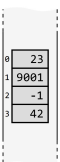


51

Sequence ADT

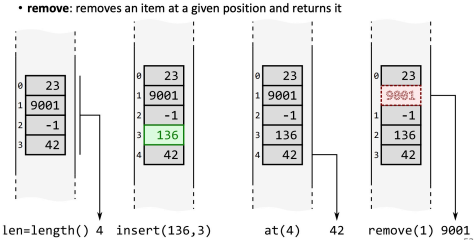
The sequence ADT is useful when you want to be able to insert, retrieve, or delete items at an arbitrary position.

The insert-at and remove-at operations change the position of items after the insertion / removal position.



Typical sequence ADT operations:

- **length**: return the number of items in the sequence
- **insert**: inserts a new item at a given position
- **at**: returns the item at a given position
- **remove**: removes an item at a given position and returns it



53

Efficiency

Algorithm step by step description of how to solve a "problem"

Time efficiency most common way to measure efficiency

Space effiency. spaces/memory requires to solve Power efficiency power consumption

Big O Notation

O(1) → constant term O(n) → linear term O(n²) → for loop (for loop)

O(logn) → a value that is * or / to reduce process O(nlogn) n times log(n)

O(n³) On · On · On O(c^n) function called multiple time.

Module 9: Dynamic Memory and ADT in C

Heap final section in C memory model
- ↳ borrow a block of memory from the heap, malloc function
  - ↳ memory allocation
- ↳ free memory → give back the memory

`p = malloc(size)` reserves `size` bytes of memory; `p` points at it.

Memory leak not free memory
- suffer degraded performance → crash
  - ↳ free or document the effect [caller must free]

Size of data type: use size-of

To create an array of 100 `int` values, you should write:

```
int *my_array = malloc(100 * sizeof(int));
```

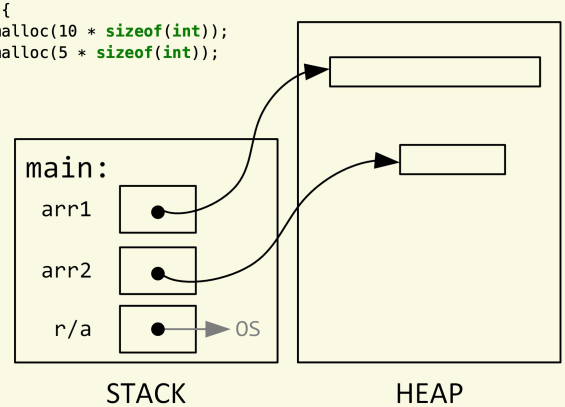
The declaration for the `malloc` function is:

```
void *malloc(size_t s);
```

The return type is a (`void *`) (*void pointer*).
This is a special pointer that can point at *any* type.

```
int *my_array = malloc(10 * sizeof(int));
struct posn *my_posn = malloc(sizeof(struct posn));
```

```
int main(void) {
  int *arr1 = malloc(10 * sizeof(int));
  int *arr2 = malloc(5 * sizeof(int));
  //...
}
```

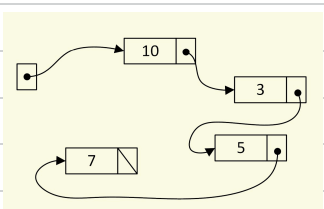


malloc (continue) unsuccessful returns NULL
↳ heap → uninitialized
once free → not able to write or read

Dynamic size of the memory to be allocated can be determined at runtime
Resizable Allocated memory can be resized
Scope Allocated memory persists until it is "freed"
A function can allocate memory that continues to be valid after the function returns.
System managed stack memory is managed by the system (stack frame)

Merge-Sort Sort the array by splitting into halves until reaching the base case, and sort.
effiency: O(nlogn)

Module 10 Linked List
Sentinel value the link in the last node (NULL)



Linked list not arranged sequentially in mem
only traverse them from front
free all node

imperative vs function imp performs mutation, function produces new value.
List shall not share nodes

wrapper approach cleaner function interfaces
reduced need for double pointers
reinforces the imperative paradigm
less susceptible to misuse and list corruptions
Ex) list

node augmentation strategy include additional info

dictionary key, value

priority queue store priority of the item

Generic Algorithm

Generic ADT does not know the type of the items

does not have any information

can not perform action